# An Efficient Digital Circuit for Implementing Sequence Alignment Algorithm

Vamsi Kundeti
(vamsik@engr.uconn.edu)
CSE Department
University of Connecticut
Storrs,CT

Yunsi Fei
(yfei@engr.uconn.edu)
ECE Department
University of Connecticut
Storrs,CT

## Abstract

The problem of Sequence Alignment (Edit Distance) between a pair of strings has been well studied in the field of computing algorithms. The classic dynamic programming-based algorithm, Needleman-Wunsch ($O(n^2)$), is widely used in practice, especially applied in the field of Biology. Biologists use this algorithm immensely to find similarities between gene sequences. Any optimization in the implementation of this algorithm will have a significant practical impact on Biological research. However, even after several decades from the time Needleman-Wunsch's algorithm was published (in 1970s), not much has been done in improving the runtime of the algorithm in real implementations. In view of this, we propose an efficient hardware implementation of the Sequence Alignment algorithm to speedup the algorithm runtime. To the best of our knowledge, this is the first hardware implementation of the Sequence Alignment algorithm. The experimental results show that our circuit implementation can achieve two orders of magintude speedup compared with the software counterpart, meanwhile reducing the area cost.

## 1 Introduction

Computing the *EditDistance* [5] between two strings is one of the most fundamental problem in computer science. Algorithms based on edit distance are used immensely in aligning biological sequences. The standard dynamic programming-based Needleman-Wunsch algorithm takes $O(n^2)$ time to compute the edit distance of two strings, $S_1 = [a_1, a_2, a_3, \ldots, a_n]$ and $S_2 = [b_1, b_2, b_3, \ldots, b_n]$, and $O(n^2)$ space to compute the actual edit script (symbolic transformation from $S_1$ to $S_2$). This algorithm has being widely used in practice, especially in biological sequence alignment. Any improvements either in time or space for the computation of this algorithm will have a significant impact on the biological research. However, in the last few decades not much has been done in improving the performance of software implementation of the algorithm. The asymptotic runtime of the algorithm remains at $O(n^2)$, although there has been some work producing a $O(\frac{n^2}{log(n)})$ algorithm [4], which has been a purely theoretical result and did not find place in real software implementations. In view of this, we propose a hardware implementation of this fundamental EditDistance algorithm which could help speeding up all sequence alignment operations. In this paper, we present an efficient circuit design for the algorithm. To the best of our knowledge, our hardware implmentation is the first one for the dynamic programming-based Sequence Alignment algorithm.

Sequence Alignment is used extensively by biologists to identify similarities between genes of different species, with genes being charecterized by DNA sequence (string of characters), e.g., $S_{dna} = [c_1, c_2, c_3, \ldots]$, $c_i \in A, T, G, C$, where $A$, $T$, $G$, and $C$ are symbols for amino acids (also called base pairs). These DNA sequences are very long, typically running into millions of base pairs. Biologists often analyze the functionality of newly discovered genes by comparing them to genes which were already discovered and whose function is fully known. Given two DNA sequences, $S_{dna}^{new}$ and $S_{dna}^{known}$, biologists perform a sequence alignment (EditDistance computation) between the two sequences to find if they have the same functionality. If the EditDistance value is below a threshold $\epsilon$, both the DNA sequences (genes) have some properties in common; otherwise they differ. BLAST (Basic Local Algorithm Search Tool) is the most popular sequence alignment tool currently used by biologists [3]. EditDistance algorithm is a fundamental building block on which the BLAST program is built, and the length of DNA sequences is a major concern for computation since the algorithm takes $O(n^2)$ time. As the size of the DNA sequences reach millions of base pairs, computing the EditDistance really suffers in performance. To overcome this problem, BLAST uses a heuristic which breaks up the long DNA sequences into smaller segments and computes the EditDistance between the broken segments and puts these segments together. As EditDistance computation between a pair of strings is a fundamental operation on which several families of sequence alignment algorithms are built, implementing this fundamental operation in hardware would speed up all the computations in software and help in scaling the software to handle longer DNA sequences in lesser time. This is the major motivation for our hardware implementation of EditDistance.

The organization of the paper is as follows. Section 2 formally defines the EditDistance problem. In Section 3, we briefly describe the dynamic programming based software algorithm. We describe our hardware implementation of RTL structures in details in Section 4. Section 5 presentes the simulation results. Finally, we draw conclusions in Section 6.

## 2 Problem Definition

Assume we have two strings, $S_1 = [a_1, a_2, a_3, \ldots, a_n]$ and $S_2 = [b_1, b_2, b_3, \ldots, b_m]$, and a set of operations $\{Insert(I), Delete(D), Change(C)\}$, each of the operations $I, D, C$ can be applied to the characters in the strings at a given position. For example, if $S_1 = [aaaabcda]$ and $S_2 = [aaabcada]$, applying an operation $D$ to the string $S_1$ at position 8 (the rightmost character) changes it to $[aaaabcd\_]$, applying an operation $I(x)$ to $S_2$ at position 8 inserts a character $x$ to $S_2$ and changes it to $[aaabcadxa]$, and applying an operation $C(b)$ to $S_1$ at position 4 which has a character $a$ makes it $[aaabbcda]$ by changing the character from $a$ to $b$. Note that each of the operations needs the position specified for operation, and $I$ and $C$ need an additional character for replacement.

The EditDistance problem asks for the minimum number of operations required to transform string $S_1$ to $S_2$ (or $S_2$ to $S_1$). A more general specification of the problem associates a cost with each of the operations, and asks for a set of operations with minimum cost which can transform $S_1$ to $S_2$. In this paper, we consider a simplified problem, where each of these operations $(I, D, C)$ has a *unit cost*, and hence minimizing the number of operations is equivalent to minimizing the cost. We consider the previous example with $S_1 = [aaaabcda]$, $S2 = [aaabcada]$, and we can transform $S_1$ to $S_2$ by a sequence of operations as follows: change character $a$ at position 4, $b$ at position 5, and $c$ at position 6 of $S_1$ to $b$, $c$ and $a$, respectively, by operations $C(b)$, $C(c)$, and $C(a)$. We can briefly describe the series of operations as a transcript $T = \{-, -, -, C(b), C(c), C(a), -, -\}$, where $-$ at positions 1, 2, 3, 7, 8 indicates no-operation, and the 3 operations at positions 4, 5, 6 result in a cost of 3 units. However, we can also transform $S_1$ to $S_2$ by applying the following transcript $T^{'}$ to $S_1$, $T^{'} = \{-, -, -, D, -, -, I(a), -, -\}$, where operation $D$ deletes the character at position 4 of $S_1$, and operation $I(a)$ inserts a character $a$ at position 7 in $S_1$. $T^{'}$ requires only 2 operations whereas $T$ requires 3 operations. The EditDistance problem asks to find the minimum number of operations which can transform $S_1$ to $S_2$.

## 3 Software Algorithm Description

The widely used algorithm for computing the edit distance of two strings is based on dynamic programming. Algorithm 1 demonstrates the pseudo-code for the algorithm of EditDistance implemented in software. We are computing an array of edit distance, D, between subsets of the two strings, i.e., $D(i, j)$ is the edit distance from the first $i$ characters of $S_1$ to the first $j$ charaters of $S_2$, and $D(n, n)$ is the minimum number of operations to transform $S_1$ to $S_2$. The first step is initialization, giving the edit distance from NULL to subsets of $S2$ ($D(0, i) = i$), and subsets of $S1$ to NULL ($D(i, 0) = i$). The computation of other array elements is performed by two loops which compute the value of $D(i, j)$ ($1 \le i \le n$,$1 \le j \le n$). It comes from the minimum one of three costs: *D(i-1,j-1)+change_cost* for changing the last character of $S_1$ when the first $i - 1$ characters of $S_1$ have been successfully transformed to the first $j - 1$ characters in $S_2$, *D(i,j-1)+insert_cost* for inserting the last character when the first $i$ characters of $S_1$ transformed to the first $j - 1$ characters of $S_2$ , and *D(i-1,j)+delete_cost* for deleting $i^{th}$ character of $S_1$ when the first $i - 1$ characters of $S_1$ have been transformed to the first $j$ characters of $S_2$. Note that the *insert_cost* and *delete_cost* are both constant unit cost, and the *change_cost* is conditionally determined by the last characters of the two strings - only when they are different, there is need for a change operation.

## 4 Hardware Implementation of the EditDistance Algorithm

In this section, we describe our efficient hardware implementation of the EditDistance algorithm.

For the core computation of $D(i, j)$ by two loops, the table $D_{n \times n}$ is filled up row by row. Let $D(i, *)$ represent the $i^{th}$ row in the table $D$. To fill up this row, we need the row of $D(i - 1, *)$ at any stage of the algorithm. Thus, a straight forward implementation may save the entire row $D(i - 1, *)$ into a register $R_1$ of size $n$, then use another register $R_2$ of size $n$ to compute $D(i, *)$. Once we are done with the $i^{th}$ row, we copy the contents of $R_2$ to $R_1$ and continue

```
INPUT    : Strings $S_1$ and $S_2$ each of size $n$
OUTPUT: Minimum number of operations to
             transform $S_1$ to $S_2$
/*Initialization*/
for $i = 1$ to $n$ do
  | $D(0, i) = i$ ;
  | $D(i, 0) = i$ ;
end
/*Recursive Computation of the Distance Table
D*/
for $i = 1$ to $n$ do
  for $j = 1$ to $n$ do
    $change\_cost = 0$;
    if $S_1[i] \neq S_2[j]$ then
      | $change\_cost = 1$;
    end
    $D(i, j) = MIN(D(i-1, j-1) +$
    $change\_cost, D(i, j-1) + 1, D(i-1, j) + 1)$
    ;
  end
end
return $D(n, n)$ ;
```

**Algorithm 1**: Pseudo-code for the algorithm of EditDistance

computation until the final row of $D$ (i.e., $D(n, *)$) is obtained. This method would require space of $2 * n$ in addition to the initialization space of $2 * n + 1$ for $D(0, *)$ and $D(*, 0)$, and also it is too complex to synthesize into hardware.

One major contribution of this paper is to propose an efficient synthesizable design, which requires only a space of $n + 2$ to compute the row $D(i, *)$ from the row $D(i - 1, *)$ at any stage of the algorithm. Our design is hierarchical and the top level block diagram is shown in Figure 1. The circuit is sequential and consists of four major blocks ComputeBlock, AlgoShifter, CounterBlock, and StringRegister. In the next several sections we describe the functionality of each of these blocks.

## 4.1 Design Block AlgoShifter

The block AlgoShifter is the core block in which all the rows $D(i, *)$ are computed efficiently during the algorithm with only a space of $n + 2$. Given two strings of length $n$, the maximum *edit distance* (cost of alignment) between them would be at most $n$. Hence, we need $log(n)$ bits to represent the *edit distance* at any stage of the algorithm, and a row of size $n + 1$, i.e., $D(i, j)$, $0 \leq j \leq n$, would need $(n + 1)log(n)$ bits (flip-flops) to represent. Figure 2 illustrates the internal

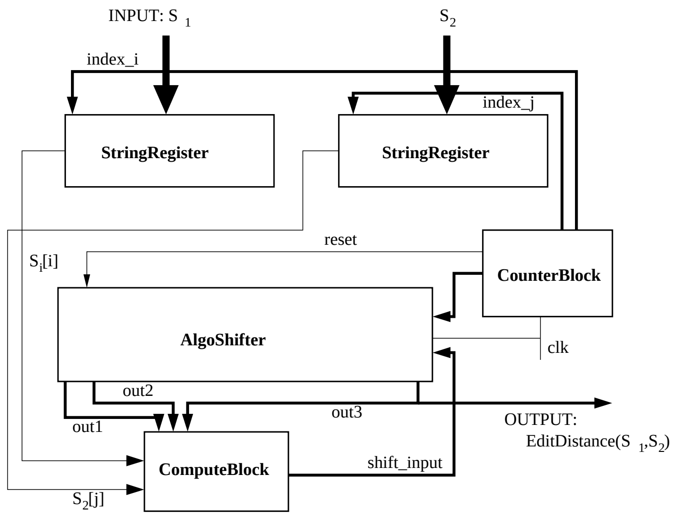

**Figure 1. Top-level block diagram of the circuit**

details of the AlgoShifter block. It contains a shift register of size $n + 2$, $S$, which has $(n + 2)log(n)$ bits in total. Let $S[i]$ denote the $i^{th}$ element that is $log(n)$ bits starting from position $ilog(n)$, $S[0]$ represent the first element, and $S[n + 1]$ the last. The block has two control input signals, *clk* and *reset*, two $log(n)$-bit data inputs, *shift_input* and *reset_input*, and three $log(n)$-bit data outputs, *out*1, *out*2, and *out*3, which represent the contents of shift register $S$ at $S[0]$, $S[1]$, and $S[n + 1]$.

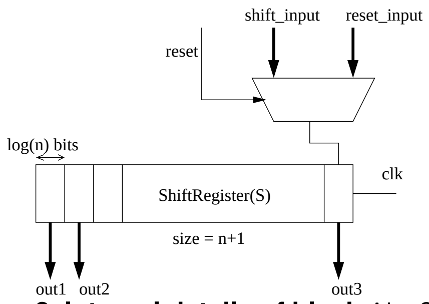

**Figure 2. Internal details of block** AlgoShifter

The block AlgoShifter performs the following functions. At every positive edge of the clock control signal (*clk*), an input between *shift_input* and *reset_input* is chosen by the multiplexer and the control signal *reset* to feed the left shifter (shift length is $log(n)$). The outputs, *out1*, *out2* and *out3*, are the values of the first, second and last elements of the shift register $S$, which are used by the ComputeBlock block, as shown in Figure 1.

We next examine how this shift register is used in implementing Algorithm 1. We take the example in which row $D(1,*)$ needs to be filled up from $D(0,*)$. The contents of $D(0,*) = [0,1,2,\ldots,n]$, and we need to compute $D(1,j)$ $(1 \leq j \leq n)$ one by one. We first check how $D(1,1)$ is computed. Three inputs are needed for the $MIN$ function in the code of Algorithm 1, which in this case are $D(0,0)$, $D(0,1)$, and $D(1,0)$. We have known $D(0,*)$ and $D(1,0) = 1$ from the initialization. We assume that the first $n$ elements of the shift register, $S[0], S[1], \ldots, S[n]$, are filled with contents from row $D(0,*)$; and $S[n+1]$ contains $D(1,0)$. With this configuration, the values of $D(0,0)$, $D(0,1)$, and $D(1,0)$ are available at $S[0]$, $S[1]$, and $S[n+1]$ to be used to compute the value of $D(1,1)$. Once $D(1,1)$ has been computed (say, with a value of $X_1$), the block proceeds to compute $D(1,2)$, which needs $D(0,1)$, $D(0,2)$ and $D(1,1)$ at this stage. It is clear that for further computation of $D(1,j)$ $(j > 2)$, the value of $D(0,0)$ is not needed any more. Thus, after each computation, we shift out the value in the shifter register which is not necessary for further computation (e.g., $D(0,0)$ - $S[0]$ in this case), and shift in the value just computed which will be needed by further computations (e.g., $D(1,1)$). In the above example, the initial content of the shift register is $S = [0,1,2,\ldots,n,1]$. After $D(1,1)$ is computed (with a value $X_1$) and a shift operation performed, $S$ becomes $[1,2,\ldots,n,1,X_1]$. After another step, $S = [2,3,\ldots,n,1,X_1,X_2]$, where $X_2$ is the computed value of $D(1,2)$. With $n$ steps of computation and shifting, $S = [n, X_1, \ldots, X_n]$, where $(X_1, X_2, X_3, \ldots, X_n)$ corresponds to $(D(1,1), D(1,2), D(1,3), \ldots, D(1,n))$. Hence, in this example, we have computed $D(1,*)$ from $D(0,*)$. We apply the same method iteratively and are able to compute all the rows in $D$.

Next, we further generalize the usage of the block. Assume that the shift register($S$) contains a row $D(i,*)$ with the configuration of $S = [D(i,0), D(i,1), D(i,2), \ldots, D(i,n), D(i+1,0)]$, after $n$ shift operations along with computations of $D(i+1,k)$ $(k \geq 1)$ as described above, the content of the shift register becomes $S = [D(i,n), D(i+1,0), D(i+1,1), D(i+1,2), \ldots, D(i+1,n)]$. Before the next row of $D(i+2,*)$ is computed, one more shift operation is needed. With $D(i+2,0)$ shifted in and $D(i,n)$ out, the three outputs of $S[0]$, $S[1]$, and $S[n+1]$ provide the values needed to compute $D(i+2,1)$, i.e., $D(i+1,0)$, $D(i+1,1)$, and $D(i+2,0)$. Since $D(i+2,0)$ is from the initialization in stead of on-the-fly computation, a control signal *reset* needs to be set. When *reset* is not set, a normal shift happens at every positive edge of *clk* to shift in the newly computed $D[i,j]$ value. Note that the control signal of *reset* is set every $n$ steps by a CounterBlock, which we describe in the next few sections. Figure 3 shows the operation of the shifter in one clock cycle.

## 4.2 Design Block ComputeBlock

At each algorithm stage, the block ComputeBlock is to compute the value of $D(i,j)$ based on three values, $D(i-1,j-1), D(i-1,j)$, and $D(i,j-1)$, which are obtained from the AlgoShifter block outputs *out1*, *out2*, and *out3*. In addition tothe inputs of $D$, the ComputBlock needs the characters in the strings at positions $i$ of $S_1$ and $j$ of $S_2$ (provided by two **StringRegister** blocks) to determine the *change_cost*. Figure 4 shows the internal details of ComputeBlock. The block is realized by using a XOR gate for the computation of *change_cost*, two adders for the operation costs for different cases, and two instances of $2-MIN\ Comparator$ which compares two inputs and outputs the minimum one.

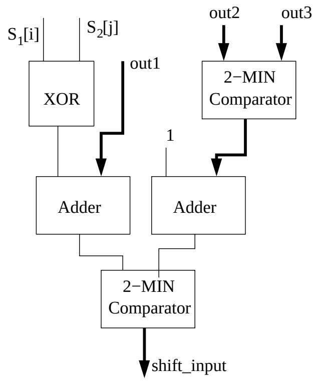

**Figure 4. Internal details of** ComputeBlock

## 4.3 Design Block StringRegister

At each computation stage, ComputeBlock requires two characters at positions $i$ and $j$ in the two strings $S_1$ and $S_2$. The StringRegister block is to take an *index* as input and outputs the character at the corresponding position in the string. This block can be realized using a N-1 multiplexer with *index* as the control input and characters in the string as the data inputs, as shown in Figure 5. The mux control input, *index*, is generated from the control block CounterBlock, which we will explain next.

## 4.4 Design Block CounterBlock

CounterBlock is the block which generates the control signals of *reset* and *index* (*index_i* for $S_1$ and *index_j* for $S_2$), and *reset_input* to the AlgoShifter block. Figure 6 shows the internal details of the CounterBlock. We implement two counters of $log(n)$ bits, $C_1$ and $C_2$,

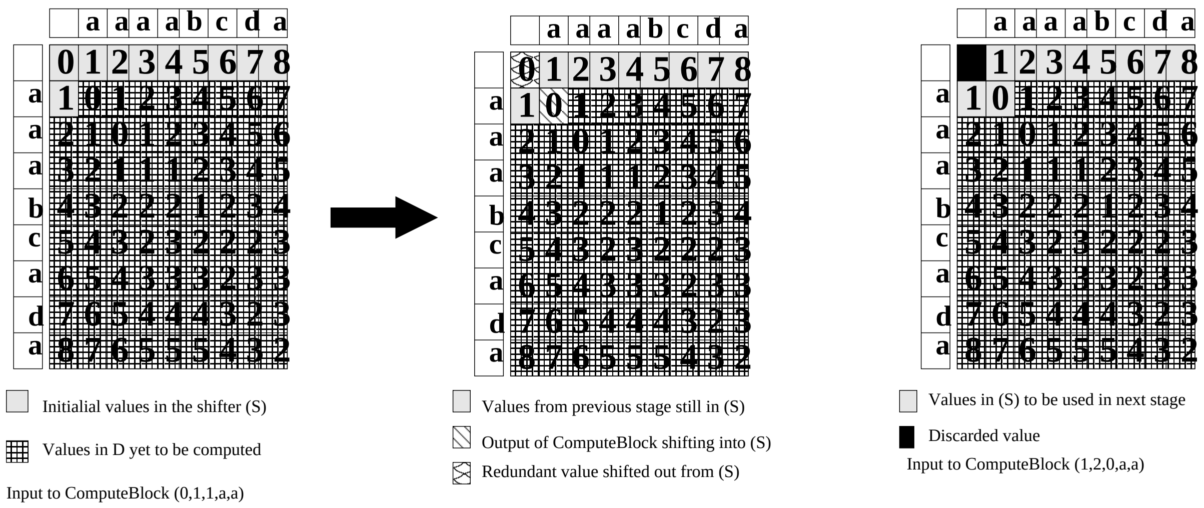
Initialial values in the shifter (S)
Values in D yet to be computed
Input to ComputeBlock (0,1,1,a,a)
Values from previous stage still in (S)
Output of ComputeBlock shifting into (S)
Redundant value shifted out from (S)
Values in (S) to be used in next stage
Discarded value
Input to ComputeBlock (1,2,0,a,a)

**Figure 3. Operation of the shifter in first clock cycle**

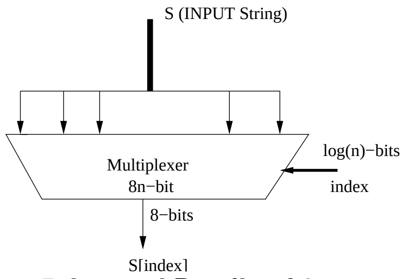

**Figure 5. Internal Details of** StringRegister

where counter $C_1$ is incremented on every positive edge of the clock with the maximum value of $n$, and $C_2$ is incremented whenever $C_1 = 0$, meanwhile, the *reset* signal is set as well. The outputs of $C_1$ and $C_2$ are for *index_i* and *index_j* respecitvely. When *reset* is set, the *reset_input* is given by $D(index\_i, 0)$.

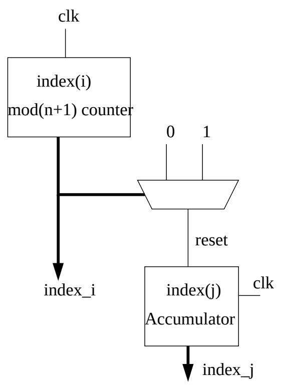

**Figure 6. Internal details of** CounterBlock

## 5 Verification and Experiments

The circuit design was implemented in Verilog and was verified using Synopsys VCS (Verilog Compiler Simulator) [2]. The complete RTL code of this design can be downloaded from the author's webpage [1]. After functionality verification, we used Cadence Encounter RTL compiler (`rc`) to synthesize the design with a TSMC 0.13um (CL013GFSG, fast.lib) standard cell library. Table1 illustrates the area and switching power with different settings of clock period (T). All our timing experiments were performed by setting input delay for all ports (except clock) to 200ps and output delay to 400ps. As the timing constraint becomes more stringent, both the area cost and dynamic power consumption are increasing. The maximum frequency of the hardware implementation is 1 GHz.

### 5.1 Speedup Estimation with Hardware Implementation

In this section, we will estimate the possible speedup in the computation with a hardware implementation of the algorithm. Without lossing generality, let $S_1$, $S_2$ be the input strings for which we need to compute the EditDistance, $|S_1| = |S_2| = n$. $M_1$ is a standard machine which does not provide any hardware implementation of the EditDistance algorithm, and $M_2$ is a machine which provides a $t \times 8$-bit hardware implementation of EditDistance algorithm. Since the input to the EditDistance problem are strings of arbitrary lengths we cannot afford to build hardware for such arbitrary lengths and hence we have to restrict for only certain fixed $t \times 8$-bit implementation in hardware. The Algorithm1 has to be modified such that it can use the $t$-bit hardware implementation available

| Timing, Area, Power | | | | |
|---|---|---|---|---|
| Clock Period(ps) | Slack(ps) | #of gates | Area | Switching Power(nW) |
| 2600 | +591 | 402 | 6551.939 | 2483438.371 |
| 2000 | +226 | 408 | 6580.787 | 3166404.936 |
| 1500 | +3 | 431 | 6694.529 | 4148261.518 |
| 1000 | +0 | 460 | 7500.792 | 7076101.428 |

**Table 1. Comparision of various design metrics with Clock Period(T)**

| Table comparing cycles required | | | | |
|---|---|---|---|---|
| Length of String | cycles on $M_1$ | $M_2$,$t = 8$ | $M_2$,$t = 16$ | $M_2$,$t = 32$ |
| 1024 | 150994944.00 | 30923764.53 | 20615843.02 | 10307921.51 |
| 2048 | 352321536.00 | 113387136.61 | 103079215.10 | 77309411.33 |
| 4096 | 1333788672.00 | 438086664.19 | 422624781.93 | 402008938.91 |
| 6144 | 3019898880.00 | 1108101562.37 | 1072023837.08 | 1041100072.55 |
| 8192 | 5184159744.00 | 1721422892.24 | 1597727834.11 | 1546188226.56 |
| 10240 | 8053063680.00 | 2602750181.38 | 2370821947.39 | 2293512536.06 |
| 14336 | 15728640000.00 | 4690104287.23 | 4303557230.59 | 4200478015.49 |
| 16384 | 38956695552.00 | 5592047419.39 | 5102421147.65 | 4870492913.66 |

**Table 2. Estimated cycles required with various hardware implementations**

on $M_2$. On machine $M_1$, the asymptotic runtime of Algorithm1 is $O(n^2)$, since Algorithm1 has to fill up the dynamic programming table $(D_{n\times n})$ element by element $(D(i,j))$. However, on machine $M_2$ which has a $t\times 8$-bit EditDistance implementation we can fill up a dynamic programming table $D'_{t\times t}$ in one hardware instruction(which takes $O(t^2)$ cycles). Algorithm 1 has to be modified to compute the table $D_{n\times n}$ block by block $(D'_{t\times t})$ rather than element by element, with the size of each block $t^2$, we will only have $\frac{n^2}{t^2}$ blocks, so the outer two loops in the modified algorithm only run $\frac{n^2}{t^2}$ times in contrast to $n^2$ times on machine $M_1$, this is an advantage on the $M_2$ because the software overhead(in maintaining the loops etc.) is now reduced by a factor of $\frac{1}{t^2}$.
We currently estimate the clock cycles required on machine $M_2$ with a $t\times 8$-bit EditDistance implementation as follows, we approximate the overhead of the software in the EditDistance to be proportional to number of cycles required, let $T_{soft}$ be the clock cycles required on machine $M_1$ then $T_{soft} = Kn^2 = (k_1 * \frac{n^2}{t^2}) * (k_2 * t^2)$ for constants $K, k_1, k_2$, from our design number of clock cycles to compute $t$-bit EditDistance is $t^2$ , the factor $(k_1 * \frac{n^2}{t^2})$ in $T_{soft}$ can be thought of contributed by the two outermost for loops running from 1 to $n$ in the increments of $t$(i.e $\{i = 1; i <= n; i+ = t\}$), from the preceeding discussion the modified algorithm which uses $t\times 8$-bit implementation also have the similar loop structure, so the number of clock cycles on $M_2$ would be $T_{hard} = (k_1 * \frac{n^2}{t^2}) * (t^2)$ and finally $speedup = \frac{T_{soft}}{T_{hard}} = k_2$. We determined $T_{hard}$ empirically finding out how much time the software takes to execute the two outer most forloops ($\{i = 1; i <= n; i+ = t\}$,$\{j = 1; j <= n; j+ = t\}$ lets call this $T_{\frac{n^2}{t^2}}$ and $T_{hard} = (T_{\frac{n^2}{t^2}}) * (\frac{t^2}{clockperiod})$. The Table2 illustrates the number of clock cycles required with $t = 8, 16, 32$, all the experiments have been performed on 2.4GHz pentium machine.

## 6 Conclusions

In this paper we have given a efficient synthesizable Design for implementing the Sequence Alignment(EditDistace) algorithm in hardware, which could speedup the computation of EditDistance.

## References

[1] Circuit Design for EditDistance. [Suppressed for paper blind review.].

[2] Synopsys Design and Simulation Tools. [`http://www.synopsys.com`].

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215:403–410, 1990.

[4] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directred graph. In *Dokl. Acad. Nauk SSSR.*, pages 487–88, 1970.

[5] C. T. H., L. C. E., R. R. L., and S. Clifford. *Introduction to Algorithms, Second Edition.* The MIT Press, September 2001.